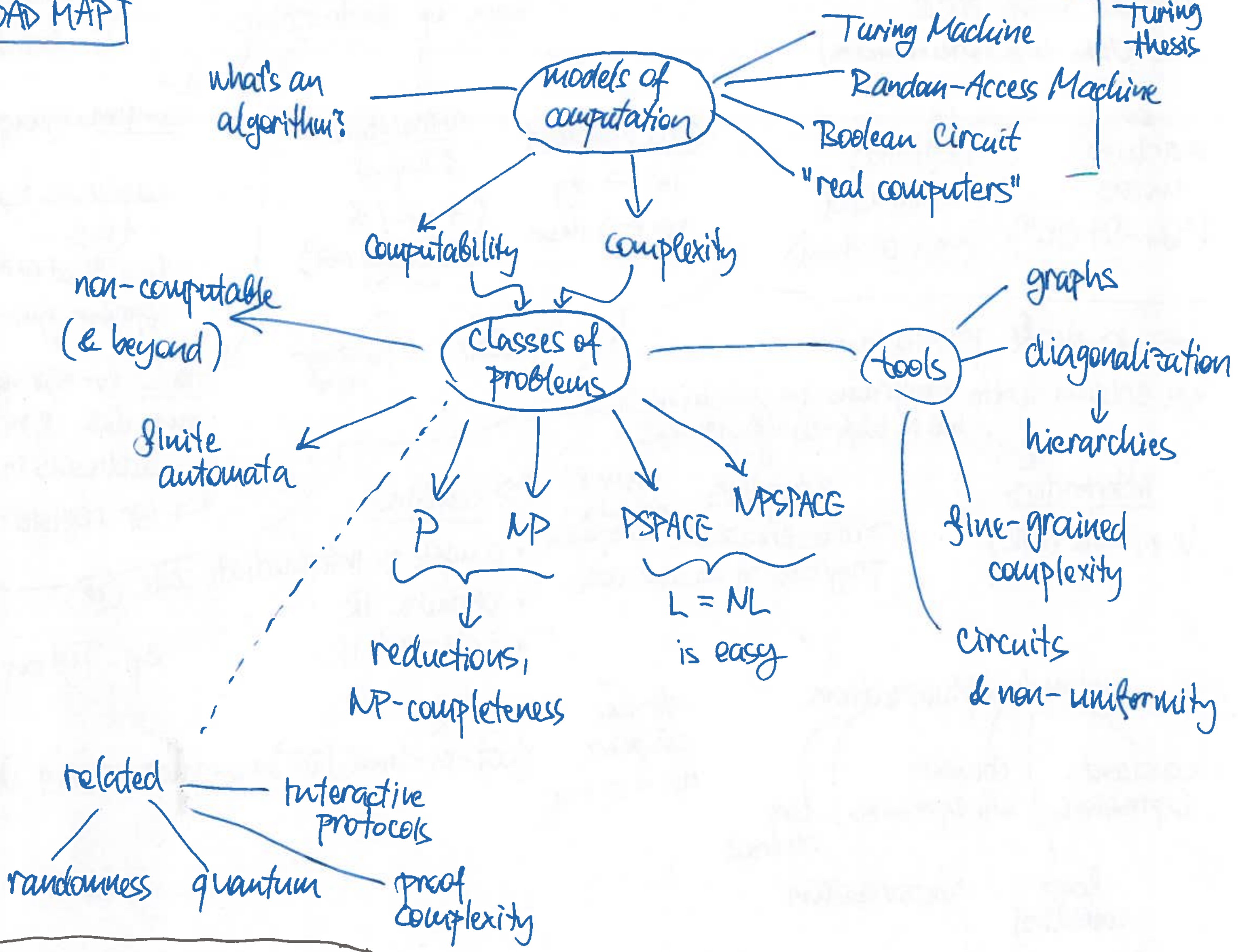
Automata & Complexity Theory

winter 2022

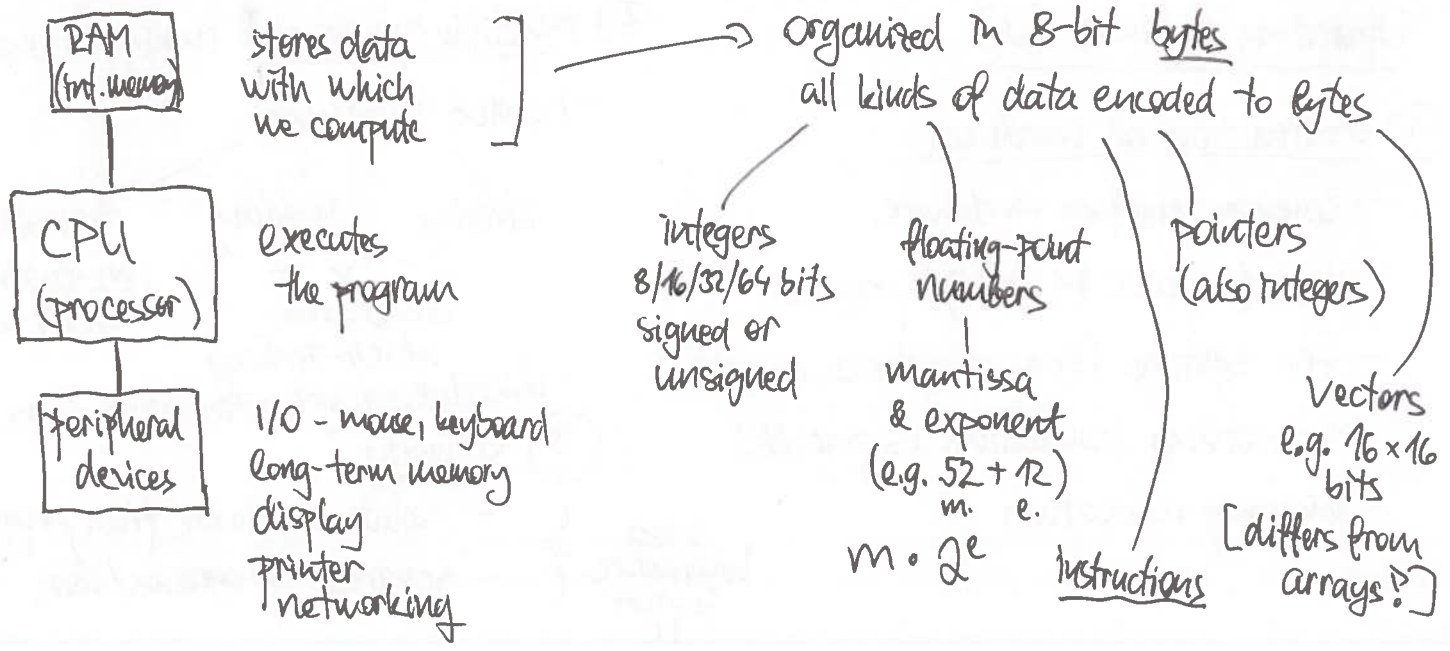
goal: build theory of computation & hardness of problems

two views:
1. an applied theory to help us use machines more efficiently
2. pure theory (branch of math / TCS)

ROAD MAP



PHYSICAL COMPUTERS & their architecture (typical case in 2022, just sketching)



machine instructions (program)
- stored in the same memory as data (Von Neumann architecture)
- can modify itself
- just another interpretation of bytes

they often work with several simple pieces of data (e.g., 64b numbers)
- machine words ("64-bit CPU")
- registers inside CPU (~10s of them)

types of instructions
- data movement
  - reg → reg
  - reg ↔ mem
- arithmetical & logical (+ - * / % AND OR XOR NOT << >>)
  - integer
  - floating-point
- stack for storing tmp. data & return addresses in RAM
  - ↳ SP register
- control (jumps)
  - → program flow, PC register
  - conditions test flags (e.g. "last arithmetical operation gave zero")

see example program...

We seldom write programs in machine insn's but in high-level languages
- interpreters (e.g., UNIX shell)
- compilers: easier & machine-independent; produce executable programs in machine code
  - automatic optimization
    - constant expressions
    - common sub-expressions
    - loop reversal
    - loop unrolling
    - vectorization
    - & many others
- combined
  - compile to intermediate code (IR)
  - optimize IR
  - interpret IR
  - e.g. Python
  - typical compilers: HLL → IR → MC
  - ↓ just-in-time (JIT) compilers – e.g. Java

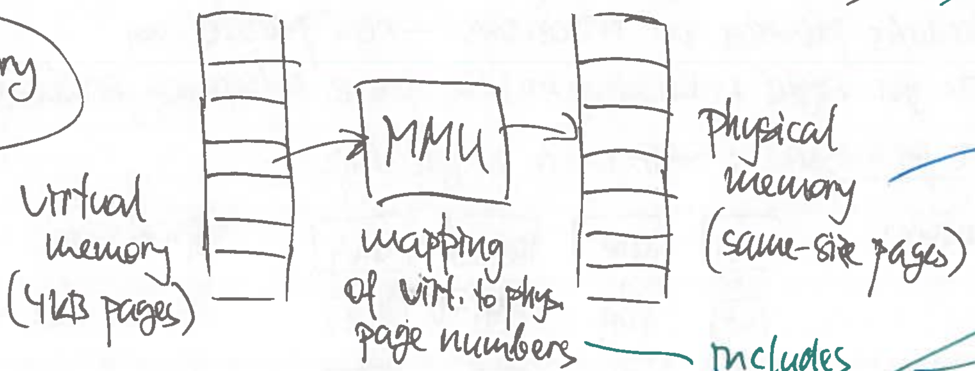
Operating systems (OS)

1. abstraction of hardware
   - common interface → drivers
   - manage access by multiple programs
   - file systems (files, directories, mounts...)
   - networking (connections vs. packets)
   - memory allocation
2. multiple processes "running at once"
   - context switching
     - sleeping, yielding: cooperative multi-tasking
     - timeslices (pre-emptive multitasking)
   - scheduling, priorities, real-time
3. security
   - isolate hardware from programs
   - separate programs/users

   need hardware support

HW features for security – privilege levels (supervisor/user mode) → system call instructions

virtual memory management


virtual memory (4kB pages)
MMU
mapping of virt. to phys. page numbers
Physical memory (same-size pages)

→ page fault exception

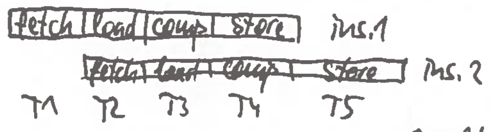
includes access rights:
- read R
- write W
- execute code X
- supervisor only (or switching tables on mode changes...)

Uses:
- protection of processes (private memory) ... RW(X) for 1 process
- shared memory ... RW(X) in multiple processes
- shared library ... R in multiple processes
- lazy allocation ... read-only shared zeroes, copy on write
- fork ... using copy-on-write mapping
- swapping ... store seldom-used pages to disk, read back on access

- caching – RAM is slow (CPU executes ~ 10^9 instructions/second, RAM latency is tens of ns) ↓ can be better only for small memory (speed of light &c.)
  - idea: small, very fast memory inside the CPU which remembers frequently used data] called a cache
  - caches 64B chunks of data (cache lines)
  - strategy:
    - write-through vs. write-back
    - when cache fills up: evict least-recently used item (LRU)
  - real caches have limited associativity → cache aliasing
  - multiple levels of caches
  - example: accessing a matrix row by row vs. column by column
    - ↳ sequential in memory
    - ↳ every access is a cache miss → very slow

  → modelling of caches, cache-oblivious algorithms (not at this lecture)

- improving execution of instructions
  - CPU works in cycles, historically 1 instruction took multiple cycles, e.g.:
    1. fetch & decode
    2. load operands
    3. compute result
    4. store result
  - pipelining



all units of CPU always busy

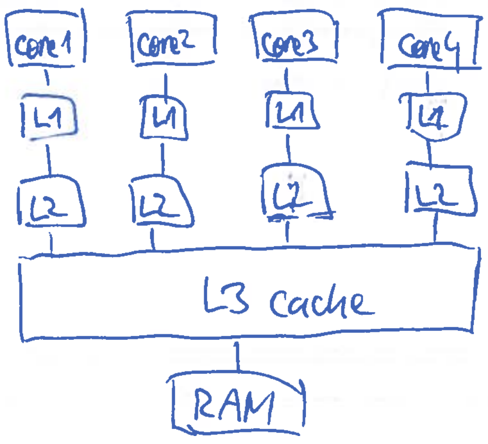
problems: dependencies & (conditional) jumps

  - superscalar CPU: multiple units for different types of instructions, can run in parallel
    → scheduling instructions to units
  - jump prediction

all this is transparent to SW (well, almost: Meltdown & other bugs)

- <u>multiple processors</u> sharing memory (SMP = Symmetric Multi-Processing)
  - OS schedules processes on processors — real parallelism
  - hard to get right: locking in SW, cache coherency protocols in HW
- <u>multi-core processors</u>: SMP on a single chip

For example:



typical sizes

32 KB code + 32 KB data

256 KB unified

8 MB unified

16 GB

- <u>multi-threaded cores</u>: two cores sharing their execution units & caches
  - unclear benefits (can even make things worse!)
- <u>virtual machines</u>: simulating a whole machine within a process
  - including supervisor mode → the VM can run its own OS
  - including virtual peripherals
  - CPUs have special support for VMs (e.g., nested paging in MMU)

<u>Relationship with theory</u>

- will ignore most machine-dependent constants
- concentrate on asymptotics ⇒ all machines (roughly) equal
- use simple mathematical machines instead
- I/O and caches need special treatment

} rest of the semester